MEASURING OPERATING SYSTEM OVERHEAD ON CMT PROCESSORS

Petar Radojković

Barcelona, January 2009.

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER of SCIENCE

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

“Којој овци своје руно смета,
Онђе није ни овце ни руна.”

Милош Војновић,
Женидба Душанова

Abstract

This thesis focuses on measuring operating system (OS) overhead on multicore multithreaded processors. Even multicore multithreaded processors are currently leading design and the tendency for the future, operating systems are still not adapted to fully utilize the potential of the novel processor microarchitecture. On the other hand, complex hardware, large number of concurrently executing processes, and new requirements like virtualization make OS have very complex role. Since OSs are not fully adapted for multicore multithreaded processors, and their role in the system expends, the overhead they introduce may be the reasons for significant performance degradation of the system.

In our study, we analyze the major sources of OS noise on a massive multithreading processor, the Sun UltraSPARC T1, running Linux and Solaris. We focus on two major sources of the OS overhead: overhead because of additional system processes running concurrently with the user application; and overhead because of virtual-to-physical memory address translation.

System processes, like interrupt handler and process scheduler, are needed in order to provide OS services to the application. Even so, they are additional processes that may interfere with user application. In our study, we quantify the overhead of interrupt handler and process scheduler of full-fledged operating systems running on Sun UltraSPARC T1 processor. We compare the results from Linux and Solaris to the ones measured in a low-overhead runtime environment, called Netra Data Plane Software Suite (Netra DPS). Netra DPS is a low-overhead environment that does not have interrupt handler nor process scheduler, what makes is a very good baseline for our analysis.

Virtual memory is a concept widely used in general purpose processors and operating systems. Through virtual memory, operating system provides the abstraction of the physical memory of the processor that significantly simplifies application programing and compiling. On the other hand, the systems that provide virtual memory, require virtual-to-physical memory translation for every instruction and data memory reference. In this thesis, we analyze the overhead of the virtual-to-physical memory translation in Linux and Solaris.

Contents

1 Introduction **1**

1.1 Introduction 1
1.2 Motivation 1
1.3 Objectives 2
1.4 Contributions 3
1.5 Organization 4

2 Multicore Multithreaded Processors **7**

2.1 Multicore processors 7
2.2 Multithreaded processors 8
2.2.1 Block multithreading 8
2.2.2 Interleaved multithreading 9
2.2.3 Simultaneous multithreading 10
2.3 Commercial multicore multithreaded processors 11
2.3.1 Homogeneous multicore multithreaded processors 11
2.3.2 Heterogeneous multicore multithreaded processors - Cell 11
2.4 Trends 12

3 Virtualization **13**

3.1 Introduction 13
3.2 Platform virtualization 14
3.2.1 Logical Domains 14
3.3 Operating-system-level virtualization 16
3.4 Application virtualization 16

4 State of the Art **17**

4.1 OS process scheduler overhead 17
4.1.1 Introduction 17
4.1.2 State of the art 17
4.1.3 Contributions 19
4.2 Memory management overhead 19
4.2.1 Introduction 19
4.2.2 State of the art 20
4.2.3 Contributions 21

5 Experimental Environment **23**
5.1 Hardware environment 23
5.2 Logical Domains 23
5.3 Netra DPS 25
5.4 Benchmarks 25
5.4.1 CPU benchmarks 26
5.4.2 Memory benchmarks 27
5.5 Methodology 27
5.6 Tools 27

6 Results and Discussion **29**
6.1 The process scheduler overhead 29
6.1.1 Process scheduler peak overhead 30
6.1.2 Process scheduler cumulative overhead 35
6.1.3 Summary 37
6.2 Overhead of the memory management 37
6.2.1 Execution time comparison 38
6.2.2 Sources of the overhead 39
6.2.3 Reducing the overhead 39
6.2.4 Summary 40

7 Conclusions **43**
7.1 Netra DPS 43
7.2 OS process scheduler overhead 43
7.3 Memory management overhead 44
7.4 Parallel applications 44

List of Figures

1.1 The OS noise effect to perfectly balanced Parallel Application 2

2.1 Different approaches possible with single-issue (scalar) processors: (a) single-threaded scalar, (b) blocked multithreading scalar, (c) interleaved multithreading scalar. 9
2.2 Simultaneous multithreading: Issuing from multiple threads in a cycle . . 9

3.1 Virtualization Concepts 14
3.2 Platform Virtualization: Sun Microsystems Logical Domains 15

5.1 LDoms setup we use in our experiments 24
5.2 Main structure of the benchmarks. The example shows the INTDIV benchmark. 26

6.1 Execution time of the INTADD benchmark when run on *strand 0* in Linux 31
6.2 Execution time of INTADD in different strands under Solaris 32
6.3 Execution time of all benchmarks running on Strand 1 under Solaris . . . 33
6.4 Execution of several INTADD repetitions with Netra DPS in strand0 . . . 34
6.5 Timer interrupt cumulative overhead in Solaris OS 35
6.6 Sample distribution in Netra DPS and Solaris 36
6.7 *Matrix by Vector Multiplication* execution time comparison 39
6.8 Effect of the page size on execution time under Solaris 40

Chapter 1
Introduction

This chapter introduces the reader to the study presented in the thesis. In Section 1.2, we describe the causes of the operating system overhead and the impact it may have to the application performance. Section 1.3 defines the objectives of our work. Section 1.4 lists the main contributions. At the end, in Section 1.5, we present the organization of the thesis.

1.1 Introduction

Modern operating systems (OSs) provide features to improve the user experience and hardware utilization. To do this, the OS abstracts real hardware, building a virtual environment, known as a virtual machine, in which the processes execute. This virtual machine makes the user's application believe it is using the whole hardware in isolation when, in fact, this hardware is shared among all processes being executed in the machine. Therefore, the OS is able to offer, through the virtual machine abstraction, features such as multitasking or a virtual extension of the available physical memory. However, these capabilities come at the cost of overhead in the application execution time.

1.2 Motivation

In fact, the overhead because of additional OS processes (e.g. interrupt handler, process scheduler, daemons) may be negligible on a single machine with few cores/threads, but may become significant for parallel applications that have to be synchronized running on a large number of cores, which is the case of High Performance Computing applications. For example, assume that a Single Program Multiple Data (SPMD) parallel application is running on a large cluster with thousands of cores. Also, in this example, assume that the application is perfectly balanced, i.e. that each process in the parallel application computes for precisely $tsec$ seconds and then communicates with other processes before starting a new iteration. In this scenario, if one of the processes in the application experiences some OS noise its iteration will require more than $tsec$ seconds. Since the other processes cannot proceed until the last task reaches the synchronization point, the whole application is slowed down as it is presented in the Figure 1.1. Moreover, as the number of cores increases, the probability that at least one process in the parallel application experiences the maximum noise during each iteration approaches 1.

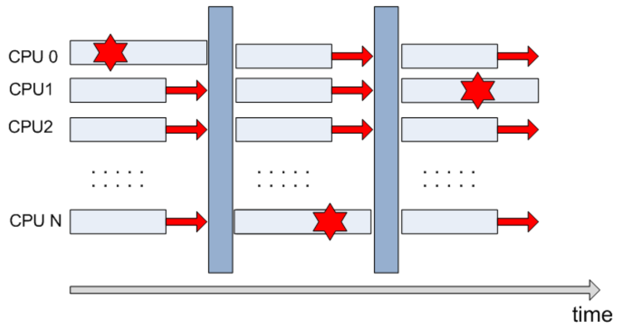


Figure 1.1. The OS noise effect to perfectly balanced Parallel Application

Operating systems use the concept of memory virtualization as a way to extend physical memory. Virtual-to-physical memory address translation is invoked on every instruction fetch and data reference. Since it requires at least one, and usually more accesses to the memory page table, it is clear that main memory access for every page table reference would significantly affect application performance.

In order to minimize page table access time, some entrances of the table can be cached in *Translation Lookaside Buffer* (TLB). The TLB is a small structure that contains the-most-probably-referenced entrances of the page table and can be quickly looked up by the memory management unit (MMU). High level of instruction level parallelism, higher clock frequencies, and the growing demands for larger working sets by applications make TLB design and implementation critical in current processors.

The memory management unit is especially sensitive to processes that use large data structures and have non-sequential access to memory. This memory behavior produces a large number of data TLB misses, which causes significant performance drop of the application.

1.3 Objectives

In our study, we analyze the major sources of OS noise on a massive multithreading processor, the Sun UltraSPARC T1 [2][4], running Solaris (version 10) [29] and Linux (Ubuntu 7.10, kernel version 2.6.22-14) [15]. We focus on two major sources of the operating system overhead; overhead because of additional system processes running concurently with the user application, and overhead because of virtual-to-physical memory address translation.

1. First, we analyze the overhead of the operating system processes. We focus on the interrupt handler and process scheduler since they cause the most performance degradation to the user applications. We measure the frequency of the interrupt handler and process scheduler are invoked and the duration of their execution. We

also measure the cumulative overhead to the user applications that is caused by repetitive execution of these system processes.

We want to distinguish two different reasons for the application slowdown. In the case user application and the system process execute on the same hardware context (i.e. *strand*), only one of the processes is able to execute at a time, while the other is stalled. Stalling the process directly affects its execution time. In the case user application and the system process execute on different strands, the reason for the slowdown is sharing hardware resources among tasks concurently executing on the processor.

2. The second goal of our study is to quantify the virtual-to-physical memory translation overhead. We focus on the penalty of main memory access in case the memory map table entry is not found in TLB. We run the experiments using different memory page size and observe the connection between memory page size, number of TLB misses, and the application performance.

We obtain the reference case running the experiments on Netra DPS, a light-weight run-time environment [8][9]. Linux and Solaris are both full OSs with many concurrent services and, since we run our experiments on a real machine, it is not easy to obtain a reference case to compare our results. A fundamental problem when determining the overhead of the OS, is that the OS noise cannot be completely removed from the system when the experiments are performed. Netra DPS is a low-overhead environment that provides less functionalities than Linux and Solaris but introduces almost no overhead. This capability makes Netra DPS a very good baseline for our analysis.

1.4 Contributions

There are three major contributions of the thesis.

1. We validate some of the well-known sources of OS noise for a chip multithreaded (CMT) processor with 32 hardware strands. We show that the process scheduler behavior in Linux and Solaris is significantly different. In Linux, the overhead is homogeneous in all hardware contexts. This is because of the fact that, in Linux, the process scheduler executes on every strand of the processor. In Solaris, the overhead depends on the particular core/strand in which the application runs. The reason for this is that Solaris binds the timer interrupt handler to the *strand 0* of the logical domain, so no clock interrupt occurs in any strand different from *strand 0*.

   We conclude that high demanding application, sensitive to the overhead introduced by the timer interrupt, running in Solaris, should not run on the first core, definitely not in the first strand. However, in the current version of Solaris, the scheduler does not take this into account when assigning a CPU to a process. Moreover, the scheduler may dynamically change the strand assigned to the application so it is up to users to explicitly bind their applications to specific strands. In our experiments, when an application is not explicitly bound to any strand, Solaris schedules it on the first strand for most of the execution, which leads to performance degradation.

2. We analyze the overhead of the memory address translation in Linux and Solaris. Our study validates number of TLB misses as one of the possible reasons for significant performance drop in the case of memory intensive applications. We also show that the number of TLB misses and the memory virtualization overhead can be greatly reduced if the memory page size is set to a proper value. However, currently it is responsibility of the user to set a proper page size to keep this overhead low. Given the importance of this feature, we advocate for a dynamic OS mechanism that should automatically set the page size in the way that fits memory requirements of application.
3. We define a framework based on the light-weight runtime environment Netra DPS to obtain a baseline execution of benchmarks without OS overhead.

1.5 Organization

Chapter 2 and Chapter 3 provide background information that will help the reader to better understand the thesis.

Chapter 2 describes multicore multithreaded processors. First, we explain the motive for the computer architecture community to move from single-threaded processors to multicore design. Later, we describe multithreaded processors with the focus on classification of multithreaded architectures and main differences among them. At the end, we present, currently, the most representative multicore multithreaded processors and the future designs.

Chapter 3 describes the main virtualization concepts. In our study, we use virtualization to run different operating systems on a single Sun UltraSPARC T1 processor.
We will pay special attention explaining *Platform virtualization*, because we directly use it to set-up the experimental environment.

Chapter 4 is the overview of the previous studies that have been exploring the operating system overhead.

Chapter 5 describes the experimental environment. We describe the processor, virtual machine, benchmarks and tools we use in the study. Since Netra DPS is not well known low-overhead environment, we explain with it with more details. Methodology is very important in OS overhead analysis. For this reason, we dedicate a section of the chapter that describes the methodology we use.

Chapter 6 presents the results of the study. We present two large sets of results: results related to process scheduler and the ones related to memory virtualization overhead. Discussion and brief summary follows each set of results.

In Chapter 7, we present the conclusions of the thesis.

Chapter 2
Multicore Multithreaded Processors

In this chapter, we briefly describe main concepts of multicore multithreaded processor design. Multicore multithreaded processors are the current trend in processor design. They are widely used in server, lap top, desktop, mobile and embedded systems. Higher throughput, simpler pipeline, lower and better distributed dissipation comparing to the single-threaded design, are only some of the reasons for their current domination on the market. Sun UltraSPARC T1, the processor we use in our study, is a multicore multithreaded processor. It contains eight identical cores each of them being multithreaded CPU.

In Section 2.1, we explain the motive for the computer architecture comunity to move from single threaded processors to multicore design. In Section 2.2, we describe multithreaded processors with the focus on classification of multithreaded architectures and main differences among them. Section 2.3 briefly describes, currently, the most representative multicore multithreaded processors. At the end, in section 2.4 we list the general trends in parallel processor architectures.

2.1 Multicore processors

In the past, the most obvious and the most simple way to increase processor performance was increasing the frequency (frequency scaling). Increasing the operating frequency, even without applying any other microarchitecture improvement, causes the instructions to execute faster that directly affects the performance. Also, from the first microprocessor till now, the technology is improving, reducing the size of single gates, that provided twice larger number of transistors on the die every in new processor generation. In past decades, additional transistors were mostly used to improve single-threaded processor performance. Every new generation of the processors had deeper and more complex pipeline, more complex prediction engines (branch prediction, value prediction), larger on-chip cache memory, etc. Even, the manufacturing technology continues to improve, still providing significantly larger number of gates in every new generation, some physical limits of semiconductor-based microelectronics have become a major design concern. There are three main obstacles for further development of single-threaded processors: *Memory Wall*, *Instruction Level Parallelism (ILP) Wall*, and *Power Wall*.

Memory Wall refers to the increasing gap between processor and memory operating frequency. Because of this, every access to the main memory will stall the application for hundreds of CPU cycles.

Instruction Level Parallelism (ILP) Wall refers to increasing difficulty to find enough parallelism in the instructions stream of a single process to keep processor cores busy. Data and control dependencies limit benefits of simultaneous instructions execution in the processor pipeline.

Power Wall refers to the increasing power dissipation and energy consumption in every new generation of the processors. The power consumption of the processor scales super-linearly with frequency increment. Even the energy consumption itself is significant problem (the energy consumption is one of the main server maintenance cost, it also increases the expenses for cooling the processor and the facility, etc.), the main problem is the power dissipation that has reached the limitation of the reliable processor operation.

Together, *Memory, ILP* and *Power Wall* combine to motivate multicore processors. Many applications are well suited to *Thread Level Parallelism* (TLP) methods, and multiple independent CPUs are commonly used to increase a system's overall TLP. A combination of increased available space due to refined manufacturing processes and the demand for increased TLP is the logic behind the creation of multicore CPUs. The fact that multicore CPUs do not require higher frequency to improve overall performance, lowers the gap between processor and memory operating frequency. Lower CPU frequency and distribution of execution units address the *Power Wall* problem.

2.2 Multithreaded processors

Multithreaded processors have hardware support to efficiently execute multiple threads. While *multicore* processors include multiple complete processing units, *multithreaded* processors try to increase utilization of a single core (single set of the processing units) by utilizing thread-level and instruction-level parallelism. Since multicore design and multithreading are two complementary techniques, frequently, they are combined in processors with several multithreading cores.

In this section, we briefly describe the main multithreading concepts: *Blocked, Interleaved* and *Simultaneous Multithreading*.

2.2.1 Block multithreading

In *Block Multithreaded* processors [41], the switch among running threads is done on the long latency event (see Figure 2.1(b)). The thread runs on the processor until it is stalled by event that causes a long latency stall (e.g. a memory reference that will access off-chip memory). Instead of waiting for the stall to resolve, a multithreading processor will switch execution to another thread that is ready to run. When the data for the stalled

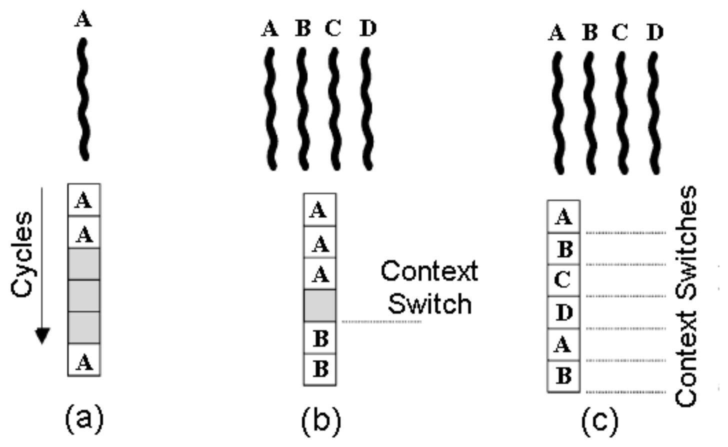


Figure 2.1. Different approaches possible with single-issue (scalar) processors: (a) single-threaded scalar, (b) blocked multithreading scalar, (c) interleaved multithreading scalar.

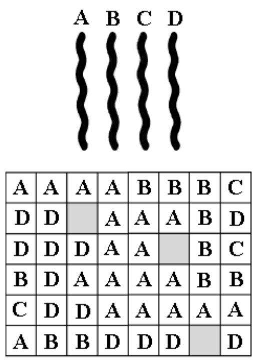


Figure 2.2. Simultaneous multithreading: Issuing from multiple threads in a cycle

thread is available, the thread will be queued in the list of ready-to-run threads. This type of multithreading is also known as *Cooperative* or *Coarse-grained* multithreading.

Many families of microcontrollers and embedded processors have multiple register banks to allow quick context switching for interrupts. Such schemes can be considered a type of *Block Multithreading* among the user program thread and the interrupt threads. *Block Multithreading* is also used in Intel Super-Threading and Itanium 2 processors.

2.2.2 Interleaved multithreading

Interleaved multithreaded processor switches threads every CPU cycle, as it is presented in Figure 2.1(c). The purpose of this type of multithreading is to remove all data dependency stalls from the execution pipeline. Since different threads are mostly independent, the probability for data dependency among instructions in the pipeline is lower (the number of the instructions from a single thread is lower comparing to the single-threaded mode execution). Initially, this way of multithreading was called *Barrel* processing. Currently, it is also referred to as *Pre-emptive*, *Fine-grained* or *Time-sliced* multithreading.

Unlike in the *Block Multithreaded* (described in the previous section), in the *Interleaved Multithreaded* processors, it is usual that more threads are being executed concurrently, each of them having instructions in the pipeline. This requires additional hardware that will track the *thread ID* of the instruction it is processing in each pipeline stage. Also, since the switch between threads is done in every CPU cycle, it is important to provide hardware support for fast context switch in the processor.

Sun UltraSPARC T1 processor is an example of *interleaved multithreaded* processor. Every core of Sun UltraSPARC T1 processor is *Interleaved multithreaded* CPU having support for concurrent execution of up to four threads. The decision from which thread the instruction will be fetched in the next cycle is determined by *Least Recently Fetched* policy among *available* threads. The thread is *not available* for the fetch if it is stalled by event that causes a long latency stall (e.g. L2 cache miss or TLB miss). As soon as the stall is resolved, the thread is again *available* for fetch.

2.2.3 Simultaneous multithreading

Simultaneous Multithreading (SMT) [41], is a technique that improves utilization of the processor resources combining superscalar execution with multithreading. In simultaneous multithreading, Figure 2.2, instructions from more than one thread can be concurrently executed in any given pipeline stage. The main change comparing to *Interleaved multithreading* is the ability to fetch and execute instructions from multiple threads at the same cycle.

Intel widely uses *simultaneous multithreading* referring to it as *Hyper-Threading* [28]. The first implementation of the *Hyper-Threading Technology* (HTT) was done on the Intel Xeon processor in 2002. Now it is available on most of Intel laptop, desktop, server, and workstation systems.

IBM includes *simultaneous multithreading* for the first time in the POWER 5 processor. IBM's implementation of simultaneous multithreading is more sophisticated, because it can assign a different priority to the various threads, is more fine-grained, and the SMT engine can be turned on and off dynamically to better execute the workloads where an SMT processor would not increase performance. The POWER5 die is consisted of two physical CPUs, each having support for two threads, what makes the total of four concurrently running logical threads.

Sun Microsystems' UltraSPARC T2 processor has eight identical cores each of them being a simultaneous multithreaded CPU.

2.3 Commercial multicore multithreaded processors

In this section, we briefly describe the most representative multicore multithreaded processors. Multicore multithreaded processors are consisted of several cores each of them being multithreaded CPU.

2.3.1 Homogeneous multicore multithreaded processors

Homogeneous multicore multithreaded processors are consisted of few copies of the CPUs (usually referred as *cores*) that are identical.

Intel *Dual-Core*[12] and *Quad-Core*[13] processors consist of two and four complete execution cores in one physical processor, respectively. *Hyper-Threading Technology* is used in each execution core, what makes the core behave as simultaneous multithreaded CPU.

IBM *POWER5* and *POWER6* processors are dual core design. Each core is capable for two way simultaneous multithreading.

Sun has produced two previous multicore processors (UltraSPARC IV and IV+), but UltraSPARC T1 is its first microprocessor that is both multicore and multithreaded. The processor is available with four, six or eight CPU cores, each core able to handle four threads concurrently. Thus the processor is capable of processing up to 32 threads at a time. Since, Sun UltraSPARC T1 is the processor we use in our study, more detailed overview of the processor is in Chapter 5. Sun UltraSPARC T2, released in 2007, is the successor of the Sun UltraSPARC T1 processor. The most important new feature is adding one more pipeline in the core what makes the core act like SMT processing unit.

2.3.2 Heterogeneous multicore multithreaded processors - Cell

The Cell Broadband Engine [24], or *Cell* as it is more commonly known, is a microprocessor designed by Sony Computer Entertainment, Toshiba, and IBM to bridge the gap between conventional desktop processors and more specialized high-performance processors, such as the NVIDIA and ATI graphics-processors. The Cell processor is consisted of two main components: the main processor called the *Power Processing Element* (PPE) and eight fully-functional co-processors called the *Synergistic Processing Elements* (SPE). The *Power Processing Element* is the IBM POWER architecture based, two-way multithreaded core acting as the controller for the eight SPEs, which handle most of the computational workload. *Synergistic Processing Elements* is a RISC processor with 128-bit *Single Instruction Multiple Data* organization designed for vectorized floating point code execution.

2.4 Trends

The general trend in parallel processor architecture development is moving from multicore dual- , tri- , quad- , eight - core chips to ones with tens or even hundreds of cores, also known as *many-core* or *massive multithreaded* processors. In addition, multicore chips mixed with multithreading and memory-on-chip show very good performance and efficiency gains, especially in processing multimedia, voice or video recognition and networking applications. There is also a trend of improving energy efficiency by focusing on performance-per-watt [11] and dynamic voltage [16] and frequency scaling [**?**]. Certainly one of the most interesting parallel processor architecture designs is forthcoming Intel architecture codenamed *Larrabee*.

Larrabee [34] is the codename for the industry's first "many-core" x86 Intel architecture. "Many-core" means it will be based on an array of many processors. The motivation for "many-core" architectures is the fact that, for highly parallel algorithms, more performance can be gained by packing multiple cores onto the die instead of increasing single stream performance.

The Larrabee architecture has a pipeline derived from the dual-issue Intel Pentium processor. The Larrabee architecture provides significant modern enhancements such as a wide vector processing unit, multi-threading, 64-bit extensions and sophisticated prefetching. The Larrabee architecture supports four execution threads per core with separate register sets per thread. This allows the use of a simple efficient in-order pipeline, but retains many of the latency-hiding benefits of more complex out-of-order pipelines when running highly parallel applications. Larrabee uses a bi-directional ring network to allow CPU cores, L2 caches and other logic blocks to communicate with each other within the chip.

The first product based on Larrabee will target the personal computer graphics market and is expected in 2009 or 2010.

Chapter 3
Virtualization

The benefit of using virtualization is significant in numerous areas of information technology. Virtualization makes it possible to achieve significantly higher resource utilization by pooling common infrastructure resources. With virtualization, the number of servers and related hardware in the data center can be reduced. This leads to reductions in real estate, power and cooling requirements, resulting in significantly lower costs. Virtualization makes it possible to eliminate planned downtime and recover quickly from unplanned outages with the ability to securely backup and migrate entire virtual environments with no interruption in service.

In our study, we use virtualization to run different operating systems on a single Sun UltraSPARC T1 processor. We use Sun Microsystems Logical Domains technology to virtually divide the processor resources in three independent environments running Linux, Solaris, and Netra DPS. In our experiments, Linux, Solaris, and Netra DPS run directly on the hardware, without any additional host OS below them.

In Section 3.1, we introduce the term *virtualization*. Section 3.2 describes *Platform virtualization* and Sun Microsystems Logical Domains virtualization technology. Since Logical Domains is the technology we use to set-up our test environment, we will explain it with more details. Later, in Section 3.3 and Section 3.4, we briefly describe the other two virtualization concepts, *operating-system-level virtualization* and *application virtualization*.

3.1 Introduction

Virtualization is a technique for hiding the physical characteristics of computing resources to simplify the way in which other systems, applications, or end users interact with those resources.

Virtualization is frequently defined as a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation and emulation [35]. Even, most of the time, it is true

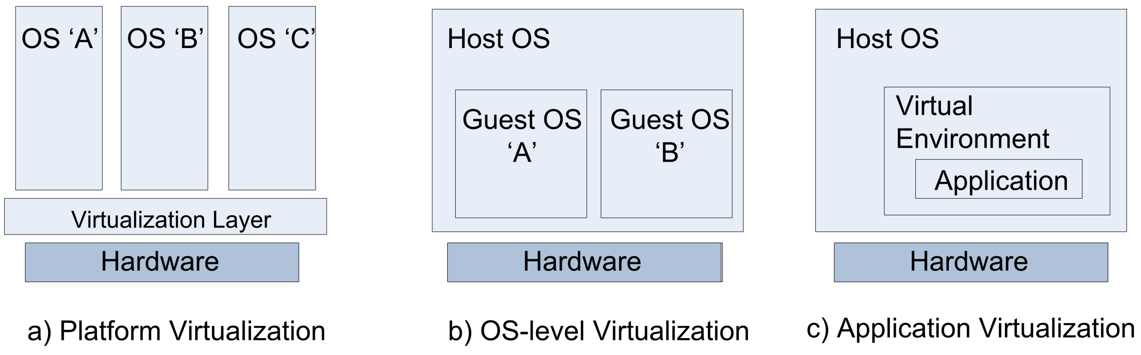


Figure 3.1. Virtualization Concepts

that virtualization implies partitioning, the same principle can be used to join distributed resources such as storage, bandwidth, CPU cycles, etc.

3.2 Platform virtualization

Platform Virtualization separates an operating system from the underlying platform resources. *Platform Virtualization* is performed on a given hardware platform by host software (a control program), which creates a virtual machine for its guest software, which is often a complete operating system (see Figure 3.1(a)). The guest software runs just as if it were installed on a stand-alone hardware platform. The guest system often requires access to specific peripheral devices (such as hard disk drive or network interface card), so the simulation must support the guest's interfaces to those devices. Even many virtual environments can be simulated on a single physical machine, this number is finite and it is limited by the amount of resources of the hardware platform.

3.2.1 Logical Domains

Sun Microsystems Logical Domains [5][6], or *LDoms*, allow the user to allocate a systems various resources, such as memory, CPUs, and I/O devices, into logical groupings and create multiple, discrete systems, each with their own operating system, resources, and identity within a single computer system (see Figure 3.2).

Hypervisor

Logical Domains technology creates multiple virtual systems by an additional software application in the firmware layer called the *hypervisor*. Hypervisor abstracts the hardware and can expose or hide various resources, allowing the creation of resource partitions that can operate as discrete systems.

The *hypervisor*, a firmware layer on the flash PROM of the motherboard, is a software layer between the operating system and the hardware. The hypervisor provides a set of

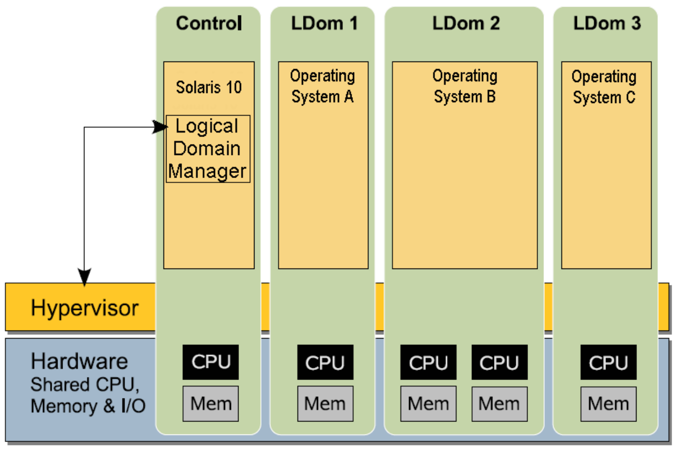


Figure 3.2. Platform Virtualization: Sun Microsystems Logical Domains

support functions to the operating system, so the OS does not need to know details of how to perform functions with the hardware. This allows the operating system to simply call the hypervisor with calls to the hardware platform. The hypervisor layer is very thin and exists only to support the operating system for hardware-specific details.

More importantly, as the hypervisor is the engine that abstracts the hardware, it can expose or hide various aspects of the hardware to the operating system. For example, the hypervisor can expose some CPUs but not others, and some amount of memory but not all, to specific operating systems. These resources can be dynamically reconfigured, which enables adding and removing resources during operation.

Logical domain

Logical domain is a full virtual machine, with a set of resources, such as a boot environment, CPU, memory, I/O devices, and ultimately, its own operating system. A logical domains (see Figure 3.2), are mutually isolated because the hardware is exposed to them through the hypervisor that virtualizes hardware resources to the upper layers. From an architectural standpoint, all domains are created equally: they are all guests of the hypervisor. Even so, they can have differing attributes that are required to perform a specific function or role.

There are several different roles for logical domains:

- **Control domain:** Creates and manages other logical domains and services by communicating with the hypervisor.
- **Service domain:** Provides services, such as a virtual network switch or a virtual disk service, to other logical domains.

- **I/O domain:** Has direct ownership of and direct access to physical input/output devices, such as a PCI Express card or a network device. Can optionally share those devices to other domains by providing services.
- **Guest domain:** Presents a virtual machine that subscribes to services provided by Service domains, and is managed by the Control domain.

A domain may have one or more roles, such as combining the functions of an I/O and service domain. In our experimental environment, Control domain also has the role of Service and I/O domain.

Other Platform virtualization suites

In addition to Logical Domains, the best known *platform virtualization* software suites are Xen [1] and VMWare Server software suite [10].

3.3 Operating-system-level virtualization

In *operating-system-level virtualization* a physical hardware platform is virtualized at the operating system level. This enables multiple isolated and secure OS virtualized environments to run on a single physical platform (see Figure 3.1(b)). The guest OS environments share the same OS as the host system i.e. the same OS kernel is used to implement all guest environments. Applications running in a given guest environment view it as a stand-alone system. The best known operating-system-level virtualization software suite is VMWare Workstation [10].

3.4 Application virtualization

Application virtualization is a software technology that encapsulates the applications from the underlying operating system and hardware on which they execute (see Figure 3.1(c)). It differs from *operating-system-level virtualization* in the sense it requires virtualization of only specific applications, instead virtualization of the whole operating system. *Application virtualization* improves portability, manageability and compatibility of applications.

The best known application virtualization software suite is *Java*. *Java* [27] is a software suite that provides a system for developing application software and deploying it in a cross-platform environment. Java programs are able to run on any platform that has a *Java virtual machine* available.

Chapter 4
State of the Art

This chapter presents the overview of the previous studies that have been exploring the operating system overhead. In Section 4.1, we describe the studies focused on the OS overhead caused by system processes like interrupt handler, daemons and process scheduler. Section 4.2 analyzes previous work done on the topic of memory virtualization overhead. At the end of each section, we emphasize the novelty and the difference of our work comparing to previous studies.

4.1 OS process scheduler overhead

4.1.1 Introduction

Modern operating systems provide features to improve the users experience and hardware utilization. One of the commonly used features is *multitasking*. *Multitasking* is a method by which multiple tasks (also known as *processes*) share common processing resources. Modern OSs provide multitasking by interleaving the execution of different tasks on the same processor. This capability offers the user the impression that several processes are executing at the same time (even with monothread architectures) and maximizes the utilization of hardware resources. To provide multitasking, the OS introduces the *process scheduler*. *Process scheduler* is responsible of selecting which process, from those ready to execute, is going to use the CPU next. Even the benefits of multitasking and other features provided by OS are evident, these capabilities come at the cost of overhead in the application execution time.

The *OS processes*, such as interrupt handler, daemons and process scheduler, that cause performance degradation of other *user process* running on the processor, are frequently called *OS noise* or *system noise*. The performance degradation because of *system noise* is very well explored in the literature. Many studies tried to quantify, characterize and reduce effects of *system noise* in application execution.

4.1.2 State of the art

Petrini et al. [32] study the influence of the system noise to hydrodynamics application SAGE [26] running at the 8,192 processor ASCI Q machine (the world's second fastest supercomputer at a time). Authors identify all sources of noise, formally categorize them, quantify the total impact of noise on application performance and determine which sources of noise have the greatest impact to performance degradation. They conclude that significant performance loss occurs when an application resonates with system noise: high-frequency, fine-grained noise affects only fine grained applications; low-frequency, coarse-grained noise affects only coarse-grained applications. Petrini et al. double SAGE's performance by eliminating the sources of system noise that have the greatest impact on performance without modifying the application itself.

The low intensity, but frequent and uncoordinated system noise causes scalability problems for fine-grained parallel (bulk-synchronous) applications. Jones et al. [23] present that synchronizing collectives consumes more than 50% of total time for typical bulk-synchronous applications when running on large number of processors. Jones et al. force simultaneous execution of daemons and tick interrupts over across the processors of an multiprocessor system that results in a speedup of over 300% on synchronizing collectives.

Tsafrir et al. [40] suggest a simple theoretical model that quantifies the effect of noise to the applications regardless the its source. The authors identify periodic OS clock interrupts (*ticks*) as the main reason for performance degradation of fine-grained applications. They also show that the indirect overhead of ticks (the cache misses they force on applications) is a major source of noise suffered by parallel fine-grained tasks. As alternative to *ticks*, Tsafrir et al. suggest *smart timers*. *Smart timers* are defined to combine accurate timing with a settable bound on maximal latency, reduced overhead by aggregating nearby events, and reduced overhead by avoiding unnecessary periodic ticks.

In his other study [39], Dan Tsafrir, compares the overhead because of ticks on two classes of applications. First, the author, explores the impact (direct and indirect) of ticks on serial application running on range of Intel platforms under 2.4.8 Linux kernel (RedHat 7.0). Later, Tsarif uses microbenchmark calibrated to executes for precisely $1ms$ in order to explore the same impact to the parallel applications. The experiments are executed on three Pentium IV generations running Linux-2.6.9 kernel. The most important contribution of this study are models that predict slowdown because of the ticks. Tsafrir presents two different models. The first model expresses the impact of ticks on serial applications. The overhead is proportional to the frequency of the clock interrupts and depends on direct and indirect tick impact. As direct impact, Tsafrir refers to the time the application is stalled because of trap and interrupt handler activities, while indirect impact can be due to evicting cache lines of the user process (that will later cause cache misses). The second model targets bulk-synchronous tasks running on large number of nodes. In this case, the overhead depends on the granularity of the task, probability that the node will be affected by noise, the latency that noise introduces into the single node execution time, and the number of nodes the application is running.

Gioiosa et al. [19] analyze the system overhead of dual AMD Opteron cluster running

Linux 2.6.5. They use the *MicroB*, the synthetic benchmark that is carefully calibrated to have constant execution time ($100\mu s$ or $10ms$) in the absence of noise. The benchmark is re-executed for $10sec$ or $100sec$ experiencing the slowdown of around 1.6% comparing to the estimated execution time. Later, the authors use the $OProfile$ tool in order to measure which and how frequently interrupting functions are called. The authors show that only few types of interrupts (global timer interrupts, local timer interrupts and network-related interrupts) constitute 95% of system noise for a wide variety of UNIX/Linux-based systems.

4.1.3 Contributions

Our contribution in the field is exploring the behavior of OS services on multicore, multithreaded processor (UltraSPARC T1) presenting ways to decrease and even completely avoid overhead due to clock tick interrupt in Solaris OS.

4.2 Memory management overhead

4.2.1 Introduction

Virtual memory is a computer system technique included in operating systems which gives an application program the impression it can use large, contiguous, non-fragmented working memory (address space).

Virtual memory divides the virtual address space of an application program into blocks of contiguous virtual memory addresses called *pages*. Data needed for translation of virtual addresses seen by the application program into physical addresses used by the hardware are located in *page table*.

Virtual to physical memory address translation is invoked on every instruction fetch and data reference. Since it requires at least one, and usually more accesses to the page table, it is clear that main memory access for every page table reference would cause significant performance drop of the application. In order to minimize page table access time, some entrances of the table can be cached. A *Translation Lookaside Buffer* (TLB) is a CPU cache that is used by memory management hardware to improve the speed of virtual address translation. The TLB is a small structure that contains the-most-probably-referenced entrances of the page table and can be quickly looked up by the memory management unit (MMU). High level of instruction level parallelism, higher clock frequencies, and the growing demands for larger working sets by applications make TLB design and implementation critical in current processors.

Some studies propose using large page size in order to increase *TLB coverage*. *TLB coverage* is the maximum amount of physical memory that can be mapped in the TLB. Two aspects of performance are affected by page size: the number of TLB misses and memory utilization. Large pages can reduce the number of TLB misses, but may also waste memory due to internal fragmentation. Small pages can increase the number of TLB misses, but use memory more efficiently since the average fragment size is smaller.

4.2.2 State of the art

Many studies [14][18][21][22] have demonstrated that performance of TLB can have notable impact on overall application performance. Anderson et al. [14] show that TLB miss is the most frequently called kernel service. Measuring large scale of applications, Huck et al. [21], demonstrate that large scale data-base intensive applications incur 5-18% overheads, pointing that extreme cases show greater than 40% TLB overhead. Kandiraju et al. [25] present an analysis of the TLB behavior for the $SPEC2000$ benchmark suite. They conclude that around one-fourth of the $SPEC2000$ applications have remarkable TLB miss rates. *Superpages* have been proposed [17][31][37][38] as a way to increase TLB coverage without increasing the number of TLB entries. *Superpages* use the same address space as conventional paging, but their size is larger than the base page size.

Romer at al. [33] analyze two aspects of performance affected by page size: the number of TLB misses and memory utilization. Large pages can reduce the number of TLB misses, but internal fragmentation can cause poor memory utilization. Small pages increase the number of misses, but use memory more efficiently. The authors propose *variable-size superpages* as the way to adapt the memory page size to the application needs in order to experience both, low number of TLB misses as well as the high memory utilization. The methodology described in the article detects when and where a superpage should be constructed based on TLB miss behavior gathered at runtime. Presented methodology is verified using ATOM, a binary rewriting tool from DEC WRL [36] that simulate the TLB behavior of the applications.

Kandiraju et al. [25] present characterization study of the TLB behavior of the $SPEC2000$ benchmark suite. Benchmarks were compiled on an Alpha 21264 machine using various C and *Fortran* compilers, and later, the simulations were running on the *SimpleScalar* toolset [1]. Experiments were conducted with different TLB configurations: the size of 64, 128, 256, and 512 entries having full-associative, 2-way and 4-way set-associative organization. Authors signify that application level (restructuring algorithms or code/data structures) or compiler directed optimizations can significantly reduce the number of TLB misses or mitigate their cost. The article also opens the discussion about the possible benefits of software directed TLB management. Changing only the replacement algorithm (adjusting it to be close to optimal), authors observe over 50% improvement in miss rates in several cases.

Talluri et al. [37] believe that increasing the *TLB coverage*, can significantly reduce the performance lost because of virtual memory translation. Being focused on the benchmarks where TLB miss handling is a significant part of the execution time, the authors

[1]The Alpha architecture simulator

prove that using *superpages* can significantly reduce the performance drop caused by TLB misses, especially in cases where large objects need to be mapped into the memory. Even so, the authors signify that using superpages requires significant operating system modifications and introduces considerable overhead. As the alternate way to improve TLB performance, Talluri et al. propose *subblock* TLB design, the concept already proposed for cache memory [20]. Authors argue that *subblocking* makes TLBs more effective than superpages while requires simpler operating system support.

4.2.3 Contributions

The topic of using large memory pages in order to decrease the number of TLB misses is widely explored. Even so, to the best of our knowledge, this is the first study that measures memory management overhead comparing execution time of the application running in OS and in environment that has simplified memory address translation, both executing on a real processor.

Chapter 5
Experimental Environment

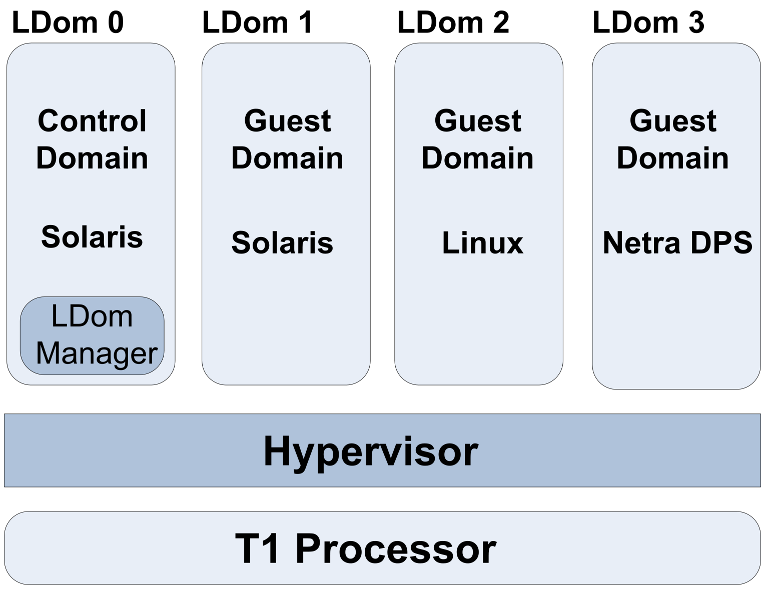
This chapter of the thesis describes the experimental environment used in the study. Section 5.1 describes the Sun UltraSPARC T1 processor. In Section 5.2, we describe Logical Domains, virtualization technology that allows us to run different operating systems on a single hardware platform. Section 5.3 describes Netra DPS, the low-overhead environment we use as a baseline in measuring the overhead of full-fledged operating systems. Benchmarks used in experiments are presented in Section 5.4. We describe methodology in Section 5.5. At the end, in Section 5.6, we describe tools used to set-up the environment parameters and to gather results.

5.1 Hardware environment

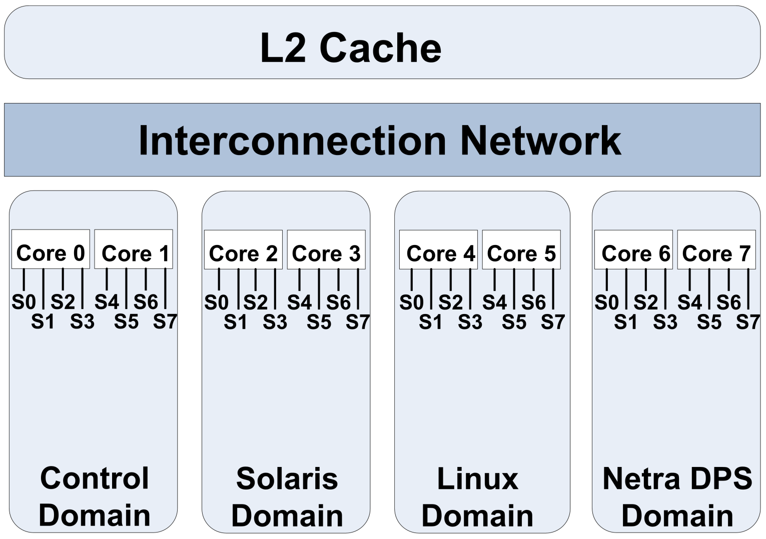
In order to run our experiments, we use a Sun UltraSPARC T1 processor running at a frequency of 1GHz, with 16GBytes of DDR-II SDRAM. The UltraSPARC T1 processor is a multithreaded multicore CPU with eight cores, each of them capable of handling four strands concurrently. Each core is a fine grained multithreaded processor, meaning that it can switch among the available threads every cycle. Even if the OS perceives the strands inside the core as individual logical processors, at a microarchitectural level they share the *pipeline*, the *Instruction* and *Data L1 Cache*, and many other hardware resources, such as the *Integer Execution Unit* or the *Frontend Floating Point Unit*. Sharing the resources may cause slower per-strand execution time but could increase the overall throughput. Beside the intra-core resources that are shared only among threads that are executed at the same core, *globally shared* resources such as *L2 cache* or *Floating Point Unit* are shared among all processes running on the processor.

5.2 Logical Domains

The Logical Domains (LDoms) technology allows a user to allocate the system's resources, such as memory, CPUs, and devices, to logical groups and to create multiple, discrete systems, each of which with its own operating system, virtual hardware resources, and identity within a single computer system. In order to achieve this functionality, we use the Sun Logical Domains software [5][6]. LDoms uses the hypervisor firmware layer of Sun CMT platforms to provide stable and low overhead virtualization. Each logical domain is allowed to observe and interact only with those machine resources that are made



(a) Logical view



(b) Mapping of the logical domains onto the cores

Figure 5.1. LDoms setup we use in our experiments

available to it by the hypervisor.

For our experimentation we create four logical domains (see Figure 5.1): one Control domain (required for handling the other virtual domains) and three guest domains running Solaris, Linux, and Netra DPS, respectively. We allocate the same amount of resources to all guest domains: two cores (8 strands) and 4 GBytes of SDRAM. For each logical domain, strand 0 (s0) is the first context of the first core, strand 1 (s1) is the second context of the first core, strand 4 (s4) is the first context of the second core, and so on.

- **Control domain**: This logical domain manages the resources given to the other domains. On this domain we install Solaris 10 (8/07).
- **Solaris domain**: This domain runs Solaris 10 (8/07).
- **Linux domain**: We run Linux Ubuntu Gutsy Gibon 7.10 (kernel version 2.6.22-14) on **Linux domain**.
- **Netra DPS domain**: On this domain we run Netra DPS version 2.0 low-overhead environment. We describe Netra DPS in Section 5.3.

5.3 Netra DPS

Netra DPS [7][9][8] is a low-overhead environment designed for Sun UltraSPARC T1 and T2 processors. Because Netra DPS introduces almost no overhead, we use it as a baseline in order to quantify the overhead of Solaris and Linux. Netra DPS introduces less overhead than full-fledged OSs because it provides less functionalities. Basically, it is used only to load the image of the code and assign hardware strands to functions.

Netra DPS does not have run-time process scheduler and performs no context switching. Mapping strands to functions is done in file before compiling the application. It is responsibility of the programmer to define the strand where the function will be executed. Netra DPS does not have interrupt handler nor daemons. The function runs to completion on the assigned strand without any interruption.

Netra DPS does not provide virtual memory abstraction to the running process and does not allow dynamic memory allocation. In UltraSPARC T1 and T2 systems, the hypervisor layer uses physical addresses (PA) while the different OSs in each logical domain view real addresses (RA). All applications that execute in Linux or Solaris OSs use virtual addresses (VA) to access memory. In Linux and Solaris, the VA is translated to RA and then to PA by TLBs and the Memory Management Unit (MMU). Applications that execute in Netra DPS environment use *real addresses* mapped in 256MB large memory pages. In the case of Netra DPS, the only address translation is from the RA into PA.

The applications for Netra DPS are coded in high level language (ANSI C) and compiled on a general-purpose operating system (Solaris 10, in our case). Later, the image that contains the application code, but also the additional information (information about mapping functions to strands and about the underlaying processor architecture) is moved to Netra DPS where it executes.

5.4 Benchmarks

We use two sets of benchmarks to test the performance of the processor. *CPU benchmarks* are simple benchmarks written in assembly we use to capture the overhead of the

	Line	Source code
	001	.inline intdiv_il, 0
	002	.label1:
B	003	sdivx %o0, %o1, %o3
O		...
D	514	sdivx %o0, %o1, %o3
Y	515	subcc %o2,1,%o2
	516	bnz .label1
	517	sdivx %o0, %o1, %o3

Figure 5.2. Main structure of the benchmarks. The example shows the INTDIV benchmark.

interrupt handler and OS process scheduler. In order to stress the memory subsystem, we create *Memory benchmarks* that use large data structures and performs significant number of non-sequential accesses to memory.

5.4.1 CPU benchmarks

Real, multi-phase, multi-threaded applications are too complex to be used as the first set of experiments because the performance of an application running on a multi-thread/core processor depends on the other processes the application is co-scheduled with. Collecting the OS noise experienced by these applications would be difficult on a real machine running a full-fledged OS. In order to measure the overhead introduced by the OS with our methodology, we need applications that have a uniform behavior so that their performance does not vary when the other applications in the same core change their phase. In order to put a constant pressure to a given processor resource we use very simple benchmarks that execute a loop whose body only contains one type of instruction. By using these benchmarks we can capture overhead due to influence of other processes running in the system, simply by measuring the benchmark's execution time.

We create a large set of benchmarks, but we present only three of them which we think are representative: integer addition (INTADD), integer multiplication (INTMUL) and integer division (INTDIV), all of them written in assembly for SPARC architectures. All three benchmarks are designed using the same principle (see Figure 5.2). The assembly code is a sequence of 512 instructions of the targeted type (lines from 3 to 514) ended with the decrement of an integer register (line 515) and a non-zero branch to the beginning of the loop (line 516). After the loop branch (line 516) we add another instruction of the targeted type (line 517) because in the UltraSPARC T1 processor the instruction after the *bnz* instruction is always executed. The assembly functions are inlined inside a C program that defines the number of *iterations* for the assembly loop. The overhead of the loop and the calling code is less than 1% (more than 99% of time proccessor executes only the desired instruction).

5.4.2 Memory benchmarks

We use a benchmark that emulates real algorithm with different phases in its execution. In particular, we build *Matrix by Vector Multiplication* benchmark that stresses the memory subsystem. For this purpose the benchmark uses large data structures and performs significant number of non-sequential accesses to memory. Thus, we try to cause significant number of data TLB misses that will slowdown the benchmark execution.

5.5 Methodology

We run each benchmark in isolation, without any other user applications running on the processor. In this way we ensure that there is no influence by any other user process and, therefore, all the overhead we detect is due to the OS activities and the activities due to maintenance of the logical domains environment that we created. To obtain reliable measurements of OS overhead, we use the FAME (FAirly MEasuring Multithreaded Architectures) methodology [42][43]. In [42][43], the authors state that the average accumulated IPC (Instructions Per Cycle) of a program is representative if it is similar to the IPC of that program when the workload reaches a steady state. The problem is that, as shown in [42][43], the workload has to run for a long time to reach this steady state. FAME determines how many times each benchmark in a multi-threaded workload has to be executed so that the difference between the obtained average IPC and the steady state IPC is below a particular threshold. This threshold is called MAIV (Maximum Allowable IPC Variation). The execution of the entire workload stops when all benchmarks have executed as many times as needed to accomplish a given MAIV value. For the experimental setup and benchmarks used in this paper, in order to accomplish a MAIV of 1%, each benchmark must be repeated at least 5 times.

The benchmarks are compiled in the Control domain using the Sun C compiler (Sun C version 5.9), and the same executables are run in Solaris guest domain. In order to run them in Linux domain, the object file obtained by the compilation in Control domain is linked with gcc (version 4.1.3) in the Linux domain.

We compile Netra DPS images in Control domain with the same Sun C compiler. To ensure the equal application behavior in the Solaris, Linux and Netra DPS domains, we use the same optimization flags in all compilations.

5.6 Tools

In order to measure the execution time of our applications, we read the *tick* register of the Sun UltraSPARC T1 processor. Reading this register returns a 63-bit value that counts strand clock cycles [3].

We use the *pmap* tool [30] to determine the size of the memory page in Solaris. The *pmap* command is used to show the individual memory mappings that make up a process address space. In order to increase the heap memory page size to 4MB and 256MB, we compile the benchmark with flags *-xpagesize_heap=4M* and *-xpagesize_heap=256M*, respectively.

In Solaris we use the *cputrack* tool [30] to determine the number of data TLB misses of applications. The *cputrack* command monitors CPU performance counters, which provides performance details for the CPU hardware caches and TLBs.

In order to determine the size of memory page in Linux, we use the *getpagesize* system call. The *getpagesize* system call invoked inside the executing code returns the memory page size in bytes.

The Solaris, Linux, and Netra DPS provide user support for binding applications to the specific hardware context (virtual processors). In Solaris, to bind process to a virtual processor we use the *processor_bind()* system call invoked in the benchmarks that we execute. The *processor_bind()* function binds a process or a set of processes defined by their *id* to a virtual processor. To bind process to a virtual processor in Linux we use the *sched_setaffinity()* function. The *sched_setaffinity()* function sets the CPU affinity mask of the process denoted by *pid*. The CPU affinity mask, in turn, defines on which of the available processors the process can be executed. In Netra DPS, binding a function to a virtual processor (strand) is done in a mapping file before compiling the application.

Chapter 6
Results and Discussion

In this chapter, we present the results of the experiments. Section 6.1 describes the results related to the process scheduler. In Section 6.2, we present the results of memory virtualization overhead. Discussion and brief summary are at the end of both sections.

6.1 The process scheduler overhead

To provide multitasking, the OS introduces the process scheduler. This scheduler is responsible for selecting which process from those ready to execute, is going to use the CPU next. To perform this selection, the process scheduler implements several scheduling policies. One of them is based on assigning a slice of CPU time, called *quantum*, to every process to delimit the period in which this process is going to be executed without interruption.

Quantum-based policies rely on the underlying hardware implementation. The hardware has to provide a way to periodically execute the process scheduler to check if the quantum of the running process has expired. To accomplish this, the current processors incorporate an internal clock that raises a hardware interrupt and allows the CPU go into kernel mode. If the quantum of the running process has expired, the process scheduler is invoked to select another task to run. In this section we will show how this hardware interrupt and the process scheduler affect the execution time of processes in Linux, Solaris, and Netra DPS.

Netra DPS applications are bound to strands at compile time and cannot migrate to other strands at run time. For this reason, Netra DPS does not provide a run time scheduler. In order to provide a fair comparison between Linux, Solaris, and Netra DPS, we decided to study the situation in which only one task is ready to execute. In this case, every time the scheduler executes, it just checks that there is no other task ready to execute in that strand. Therefore, the overhead we report concerning the process scheduler is the lowest that can be observed.

Moreover, having more than one application running at the same time will make the study more complex to analyze, as the overhead of the OS on one application could overlap with the influence of the other running applications.

6.1.1 Process scheduler peak overhead

In order to measure the influence of the process scheduler, we consecutively execute 1000 *repetitions* of every benchmark, where each repetition lasts approximately $100\mu s$. The results obtained are the following:

Linux

Figure 6.1 shows the execution time per repetition of the INTADD benchmark in Linux when it is bound to *strand 0*. In Figure 6.1, the X-axis shows the time at which each repetition starts and the Y-axis shows the execution time of the repetition. We observe that the average execution time of repetition is $100\mu s$. The important point in this figure is the presence of periodic noise. This noise occurs every 4 milliseconds (250Hz frequency) and corresponds to the interrupt handler associated to the clock tick. Since Linux implements a quantum-based scheduling policy (quantum scheduler with priority), the process scheduler has to be executed periodically to check if the quantum of the process currently being executed is expired or not, or if a higher priority process has waken up. Hence, even if INTADD is executed alone in the machine, its execution is disturbed by the interrupt handler. This makes some repetitions of the benchmark running longer ($123\mu s$), which represents a slowdown of 23%. We repeat the experiment for INTMUL and INTDIV benchmarks. The results are the same as for INTADD benchmark; every $4ms$ we detect the repetitions that have longer execution time.

We re-execute the INTADD benchmark in other strands of the processor and obtain the same behavior. In fact, those peaks appear regardless of the strand in which we run the benchmark. This is due to the fact that, in Linux, in order to provide scalability in multithreaded, multicore architectures, the process scheduler is executed in every strand of the processor.

Solaris

Solaris behaves different then Linux. Figure 6.2 shows the execution time of the INTADD benchmark when it is executed in Solaris. In this case, INTADD is statically bound to *strand 0* (Figure 6.2(a)), *strand 1* (Figure 6.2(b)) and *strand 4* (Figure 6.2(c)).

Figure 6.2(a) shows that, when the benchmark runs in *strand 0*, the behavior is similar as in Linux. The reason is the same. Since Solaris provides a quantum-base selection policy, the clock interrupt is raised periodically. But, in this case, the frequency of the clock interrupt is 100Hz.

Figure 6.2(b) shows execution time of INTADD benchmark when it is bound to *strand 1*, a strand on the same core where the timer interrupt handler runs. In this case, the peaks are smaller since they are the consequence of sharing hardware resources between two processes running on the same core and not due the fact that the benchmark is stopped because execution of the interrupt handler and the process scheduler, as it is in the case in *strand 0*. In Linux we do not detect similar behavior[1], because the impact is hidden by

[1]Peaks in benchmark execution time because of sharing hardware resources with the interrupt handler and the process scheduler

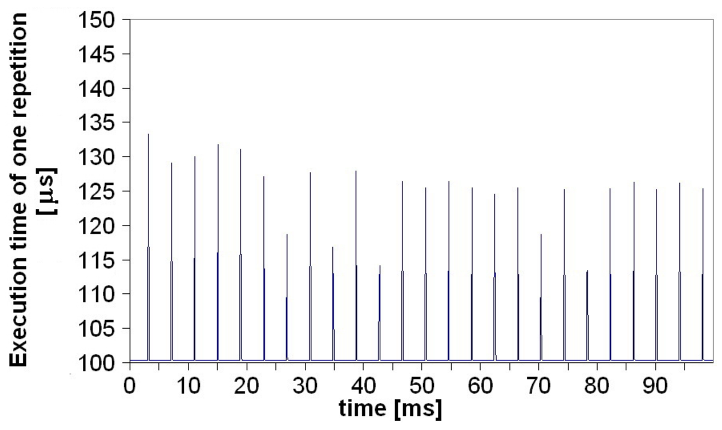


Figure 6.1. Execution time of the INTADD benchmark when run on *strand 0* in Linux

execution of timer interrupt routine on each strand.

In the UltraSPARC T1 processor, all strands executing in the same core share the resources of the core. One of the resources is *Instruction Fetch Unit*. Even if two or more threads are ready to fetch an instruction, only one of them[2] is able to do it in the next cycle[3]. Instruction fetch of other threads is delayed for the following cycles. As a consequence, when INTADD runs in *strand 1*, and no other thread is executed in any other strand in the core, it is able to fetch instruction in every cycle. But, when the clock interrupt is raised, and the interrupt handler is executed, the IFU is shared among both processes what sometimes makes INTADD to be delayed because IFU is assigned to the interrupt handler. This makes INTADD suffer some performance degradation.

When INTADD executes in *strand 4* we do not detect any peaks, see Figure 6.2(c). Since Solaris binds the timer interrupt handler to *strand 0*, no clock interrupt is raised in any strand different from *strand 0*. For this reason, *strand 4* (nor any other strand on the same core) does not receive any clock interrupt, which makes the behavior of INTADD stable.

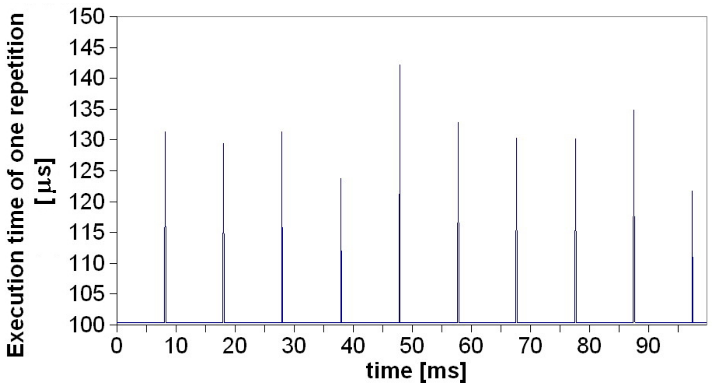
We repeat the experiment for the INTMUL and INTDIV benchmarks. When the tests are performed on *strand 0*, we detect the same overhead in execution time (15μs to 45μs over the overall behavior) with the same tick frequency (100Hz). In addition, when the benchmarks are executed in *strand 4*, the peaks also disappear as it happens to INTADD. But, when the experiments are executed on *strand 1*, as shown in Figure 6.3, we notice some differences with respect to the execution time of INTADD (Figure 6.3(a)), INTMUL (Figure 6.3(b)) and INTDIV (Figure 6.3(c)). Note that the scale of Figure 6.2(c) is different.

In order to clarify this point, we run in Solaris 50,000 repetitions of every benchmark (INTADD, INTMUL, INTDIV) on *strand 0* and *strand 1*. The results are summarized in Table 6.1.

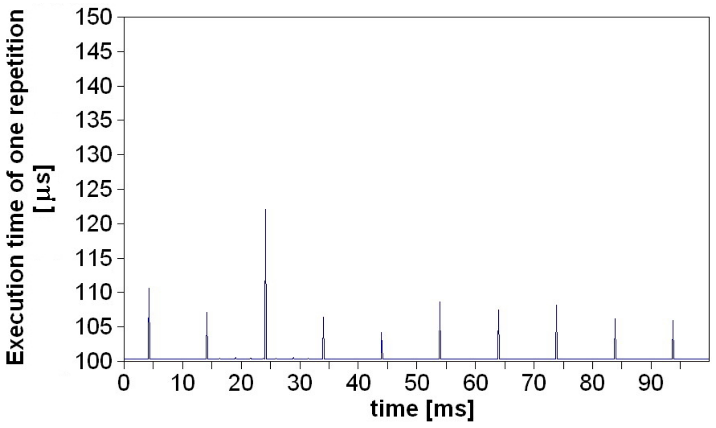
We observe that the average overhead is almost the same for all three benchmarks (with

[2]In the case of the UltraSPARC T1 processor, *Least Recently Fetched* fetch policy determines which among available threads will access IFU next.

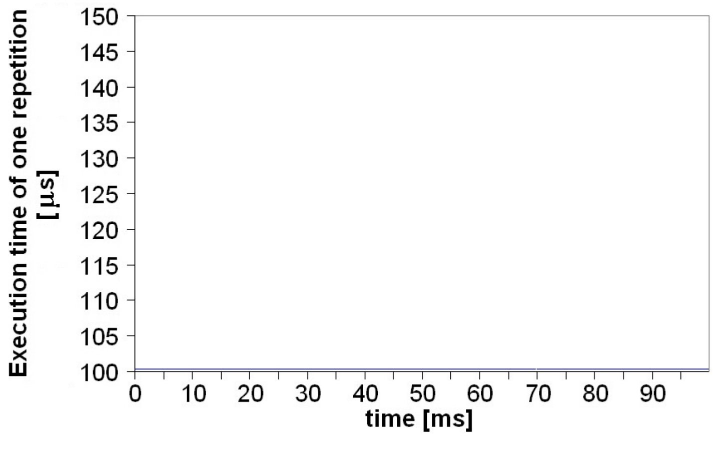
[3]*Instruction Fetch Unit (IFU)* is able to handle up to one instruction fetch per cycle



(a) Strand 0 (Core 0)

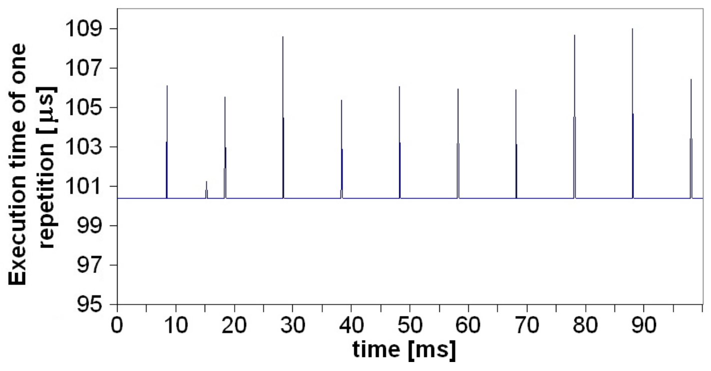


(b) Strand 1 (Core 0)

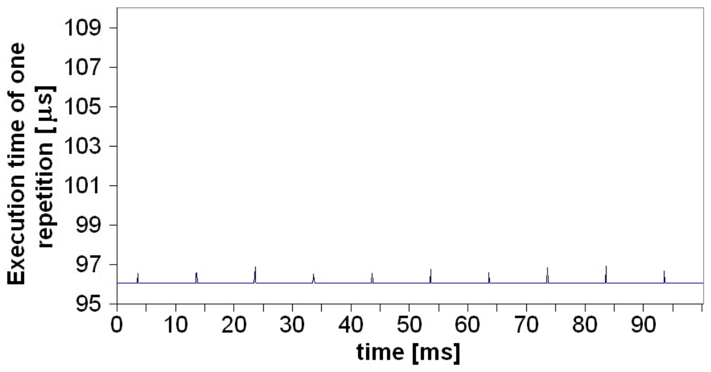


(c) Strand 4 (First strand on Core 1)

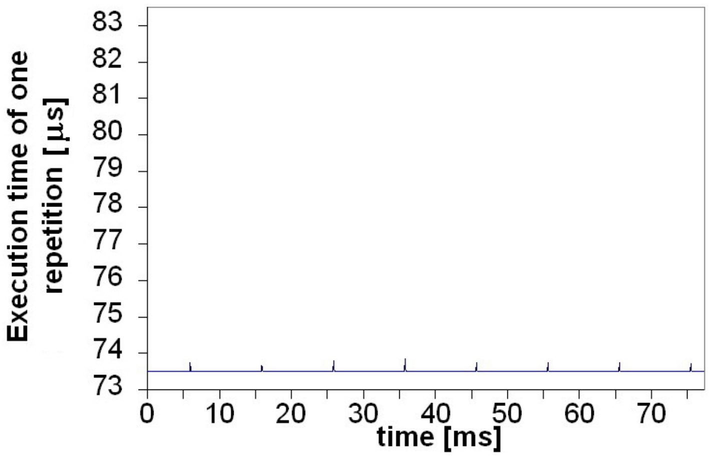
Figure 6.2. Execution time of INTADD in different strands under Solaris



(a) INTADD executed in Strand 1



(b) INTMUL executed in Strand 1



(c) INTDIV executed in Strand 1

Figure 6.3. Execution time of all benchmarks running on Strand 1 under Solaris

a small, 2.2% difference in the worst case) when we run them in *strand 0*. In this case the overhead is introduced because the benchmark is stopped and the interrupt handler

Benchmark	CPI	Avg. overhead [μs]	
		Solaris - strand 0	Solaris - strand 1
INTADD	1	26.415	6.528
INTMUL	11	26.657	1.195
INTDIV	72	26.218	0.823

Table 6.1. Average time overhead due clock tick interrupt

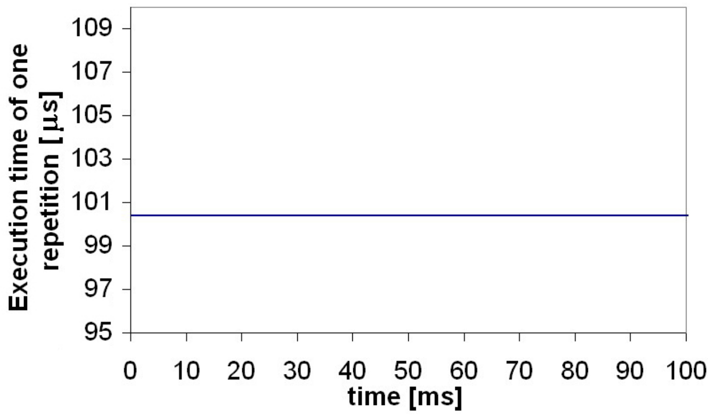


Figure 6.4. Execution of several INTADD repetitions with Netra DPS in strand0

and the OS process scheduler run in *strand 0*. The overhead is different when we execute threads in *strand 1*. In this case, the overhead is due to the fact that the benchmark running on *strand 1* shares the fetch unit with the timer interrupt handler and process scheduler when they run on *strand 0*. Given that the pressure to the instruction fetch unit depends on CPI of an application, the overhead is different for each benchmark. In fact, the lower the CPI of a benchmark, the higher is the pressure to the fetch unit, and the higher the effect it suffers when an additional process runs on the same core.

Netra DPS

Finally, when the INTADD benchmark is executed in Netra DPS, as shown in Figure 6.4, the peaks do not appear. This is due to the fact that Netra DPS does not provide a run-time scheduler. Threads executed in this environment are statically assigned to hardware strands during compilation. At runtime, threads run to the completion on one strand, so no context switch occurs. In Netra DPS, ticks are not needed for process scheduling which removes the overhead from the benchmark execution. This behavior is present in every strand assigned to Netra DPS.

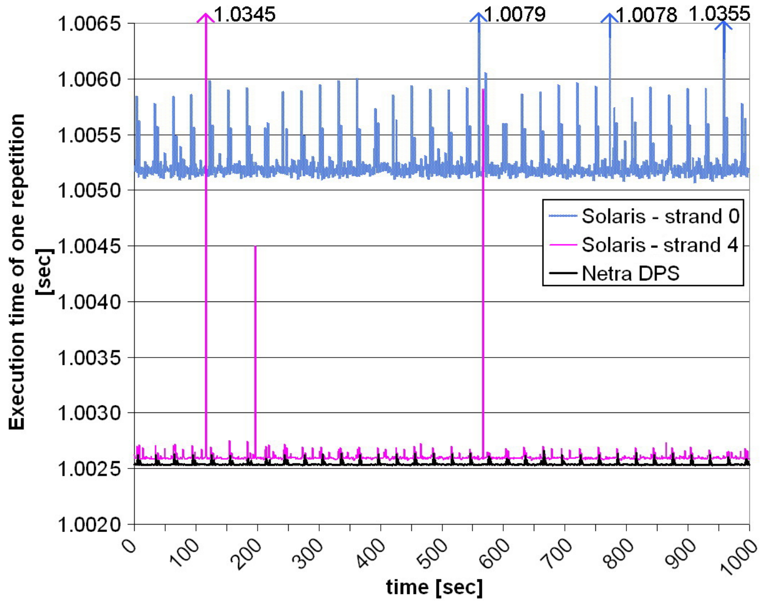


Figure 6.5. Timer interrupt cumulative overhead in Solaris OS

6.1.2 Process scheduler cumulative overhead

From the previous section it may seem that the overhead of the OS on the average is small since it only affects few repetitions of the benchmark execution. In fact, process scheduler overhead can only be detected when measurements are taken at a very fine grain, as in the previous examples. But it is important to notice that, when moving to a larger scale, even if no overhead coming from the scheduler can be detected, this overhead accumulates in the overall execution time of the benchmarks. To show this effect, we repeat the experiments but extending the total execution time of each repetition of the benchmarks to 1 second. We make this experiment in Netra DPS and Solaris, running benchmarks on both *strand 0* and *strand 4*.

Figure 6.5 presents the behavior of the INTADD benchmark. In this Figure, the bottom line corresponds to Netra DPS, whereas the middle and the topmost lines correspond to the benchmark when it is executed with Solaris in *strand 4* and *strand 0*, respectively. The X-axis shows the time at which each repetition starts and the Y-axis describes execution time per repetition.

As shown in Figure 6.5, Netra DPS, for the reasons explained in the previous section, is the environment that presents the best execution time for the benchmark even when measurements are taken in coarse grain. Small peaks appearing in the execution of INTADD under Netra DPS come from some machine maintenance activities due to the Logical Domain manager. Unfortunately, this overhead noise cannot be evicted when other logical domains (Control domain, Linux, and Solaris domains) are present on the machine. Our experiments reveal that maintenance of logical domains causes a global overhead noise in all strands of the processor similar to those shown in Figure 6.5 for Netra DPS. In order to confirm that those peaks are neither due to execution application in Netra DPS nor the

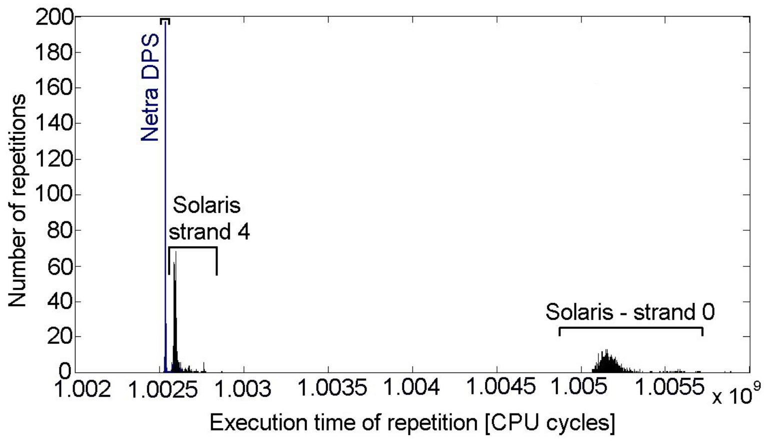


Figure 6.6. Sample distribution in Netra DPS and Solaris

hypervisor activities, we re-execute benchmarks in Netra DPS without LDoms and we detect no peaks in execution time.

The second best time in Figure 6.5 relates to the execution of the benchmark with Solaris in *strand 4*. Notice that the overhead peaks (the smallest ones) caused by the LDom management layer are also present.

Finally, the benchmark presents its worst execution time when it is executed with Solaris in *strand 0* (topmost line in Figure 6.5). This overhead comes from the cumulation of the clock interrupt overheads.

Figure 6.6 draws the distribution of the samples (for Netra DPS, Solaris-strand 4 and Solaris-strand 0) shown previously in Figure 6.5. For Figure 6.6, the X-axis describes execution time, whereas the Y-axis shows the number of samples (repetitions) that have a given execution time. In this Figure, samples make three groups from right to left. The first group, ranging from $1.005 * 10^9$ to $1.006 * 10^9$ cycles, covers the samples of the execution of the INTADD benchmark with Solaris in *strand 0*. The second group, from $1.0026 * 10^9$ to $1.0028 * 10^9$, is related to the execution with Solaris in *strand 4*. And, finally, the third group corresponding to Netra DPS is centered in the execution time point of $1.0025 * 10^9$ cycles.

Two major conclusions can be drawn from Figure 6.6. First, as previously seen in Figure 6.5, Netra DPS is the configuration that presents the smallest variance in the execution of all repetitions. All repetitions last for $1.0025 * 10^9$ cycles. Second, Solaris in both *strand 0* and *strand 4* presents higher variance. The range of variation is on average $0.0001 * 10^9$ and $0.003 * 10^9$ cycles when a benchmark runs on *strand 4* and *strand 0*, respectively.

Figure 6.5 and Figure 6.6 lead us to the conclusion that Netra DPS is a very good candidate to be taken as a baseline for measuring the overhead of operating systems since it is the environment that clearly exhibits the best and most stable benchmark execution time.

Stable execution time makes Netra DPS an ideal environment for parallel applications running on large number of cores, as it is in the case of HPC applications.

6.1.3 Summary

We show that the process scheduler behavior in Linux and Solaris is significantly different. While in Linux the overhead is homogeneous in all strands, in Solaris the overhead depends on the particular core/strand in which the application runs.

When we execute our benchmarks in Linux, we detect periodic overhead peaks with a frequency of 250Hz, which corresponds to timer interrupt handler. We re-execute the benchmarks in different strands of the processor, obtaining the same behavior. This is due to the fact that in Linux the process scheduler executes on every strand of the processor.

In Solaris, we detect different performance overhead depending on the strand a benchmark executes:

- When an application runs in *strand 0* we observe the highest overhead, regardless of the type of instructions the application executes.
- When the application runs in the same core with the timer interrupt handler, but on the strand different from *strand 0*, we also observe some smaller overhead the intensity of which depends on the application's CPI (Cycles Per Instruction): the higher the CPI, the higher the overhead experimented by the application.
- We detect no timer interrupt overhead when applications execute on a core different than the one on which the timer interrupt handler runs.

The reason for this is that Solaris binds the timer interrupt handler to the *strand 0* of the logical domain, so no clock interrupt occurs in any strand different from *strand 0*.
Hence, high demanding application, sensitive to the overhead introduced by the timer interrupt, running in Solaris, should not run on the first core, definitely not in the *strand 0*. However, in the current version of Solaris, the scheduler does not take this into account when assigning a CPU to a process. Moreover, the scheduler may dynamically change the strand assigned to the application so it is up to users to explicitly bind their applications to specific strands. In our experiments, when an application is not explicitly bound to any strand, Solaris schedules it on the first strand for most of the execution, which leads to performance degradation.

6.2 Overhead of the memory management

Modern OSs use the concept of memory virtualization as a way to extend physical memory. In order to make translation from virtual to physical memory address, memory virtualization requires access to memory map table (located in main memory) and TLB hardware. In the event of an entry of memory map table needed for translation is not located in TLB, a TLB miss happens. Resolving a TLB miss requires access to main memory introducing overhead in process execution. UltraSPARC T1 uses a 64 entries Data TLB (DTLB) per core that is shared between the four threads running in the core. The translation table entries of each thread are kept mutually exclusive from the entries of the other threads. The memory management unit is especially sensitive to processes that use large data structures and have non-sequential access to memory. This memory behavior produces a large number of data TLB misses.

In UltraSPARC T1 systems, the hypervisor layer uses physical addresses (PA) while the different OSs in each logical domain view real addresses (RA). All applications that execute in Linux or Solaris OSs use virtual addresses (VA) to access memory. The VA is translated to RA and then to PA by TLBs and the Memory Management Unit (MMU).

Applications that execute in Netra DPS environment use *real addresses* mapped in 256MB large memory pages. In the case of Netra DPS, the only address translation is from the RA into PA.

The translation from RA to PA is present in all logical domains and the overhead is the same in all cases. The cause for the performance difference is virtual-to-real address translation, as it is different for full-fledged OSs (Linux and Solaris) and Netra DPS.

In this section, we compare the execution time of *Matrix by Vector Multiplication* benchmark running on the Linux, Solaris and Netra DPS logical domains with different compiler optimizations. After that, using the multiple page size support provided in Solaris OS, we execute the same benchmark in Solaris and we force OS to use large, 4MB and 256MB, page size. At the end, we connect speedup obtained using large page sizes to decreased number of data TLB misses.

6.2.1 Execution time comparison

We use two levels of compiler optimization to test the effect the automatic optimization may have on the memory access and hence on overhead caused by OS memory management. Figure 6.7 shows the execution time (in seconds) of the *Matrix by Vector Multiplication* benchmark when it is compiled with different optimization levels and run in different OSs. The left group of bars shows, the execution time when compiler optimization $-O0$ is applied. The right group of bars shows the same when we use $-O3$ compiler optimization. We observe that the code executes faster in Netra DPS domain (this will be used as a baseline below) than in Solaris and Linux.

The absolute overhead introduced by the memory management does not change when the optimization level is changed; it is $13sec$ for Solaris and $19sec$ for Linux. Since total execution time when benchmark is compiled with $-O3$ flag is less, the relative speedup is higher for $-O3$ optimization level.

When running on Linux domain execution of the application is 69.86% slower for the code compiled with $-O0$ and 155.16% slower for $-O3$. Execution time when the application is running on Solaris domain is 44.64% and 98.07% larger when code is compiled with $-O0$ and $-O3$ flags, respectively.

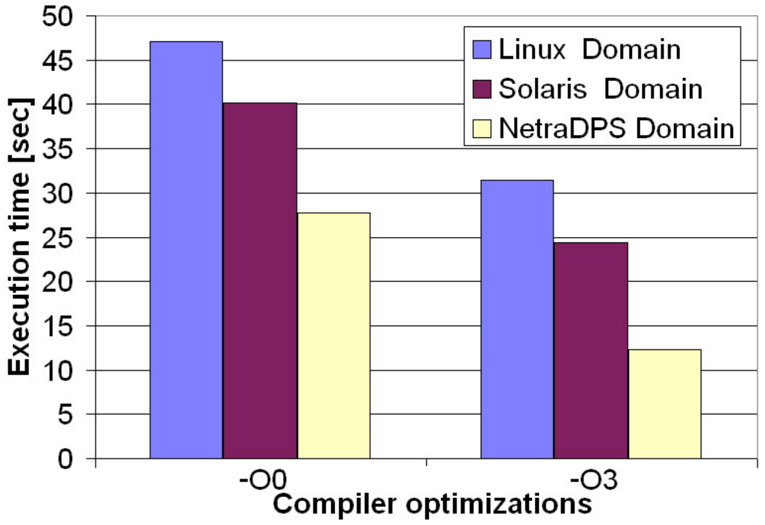


Figure 6.7. *Matrix by Vector Multiplication* **execution time comparison**

6.2.2 Sources of the overhead

The main reason behind this significant slowdown when application runs in Linux and Solaris OS resides in the memory management. The default memory page size in Linux and Solaris is 8KB and 64KB, respectively. Using small memory pages requires large number of entries in memory map table, that do not fit in TLB. As a consequence, a lot of TLB misses are expected. Netra DPS environment uses 256MB large memory pages in order to increase the TLB coverage.

Table 6.2 shows the number of data TLB misses for different page sizes and different compiler optimization levels. The first column presents, from top to down, the number of data TLB misses for the default, 4MB and 256MB heap page size, for compiler optimization $-O0$. The second column presents the same results, but this time with compiler optimization $-O3$. As it is seen in Table 6.2, execution of benchmark in Solaris with default page size causes significant number of data TLB misses and precisely resolving those misses introduces overhead in execution time.

We observe that the number of data TLB misses, for the same memory page size, is almost the same regardless of the compiler optimizations uses. This matches with the same absolute speedup variance seen in Figure 6.7.

6.2.3 Reducing the overhead

In a second set of experiments, we use Solaris support for multiple page sizes in order to decrease the number of data TLB misses and improve benchmark performance. Given that 64KB page size causes large number of data TLB misses, we increase the page size to 4MB and 256MB for heap memory.

Figure 6.8 shows the execution time of the *Matrix by Vector Multiplication* benchmark running in Solaris when it is compiled with different optimization levels and different memory heap page sizes. We observe that when we use the $-OO$ compiler optimization

	$-O0$	$-O3$
Default Page Size [64KB]	61,091,463	61,092,618
4MB Page Size	947,825	947,206
256MB Page Size	12,650	12,626

Table 6.2. Number of data TLB misses for different page size in Solaris

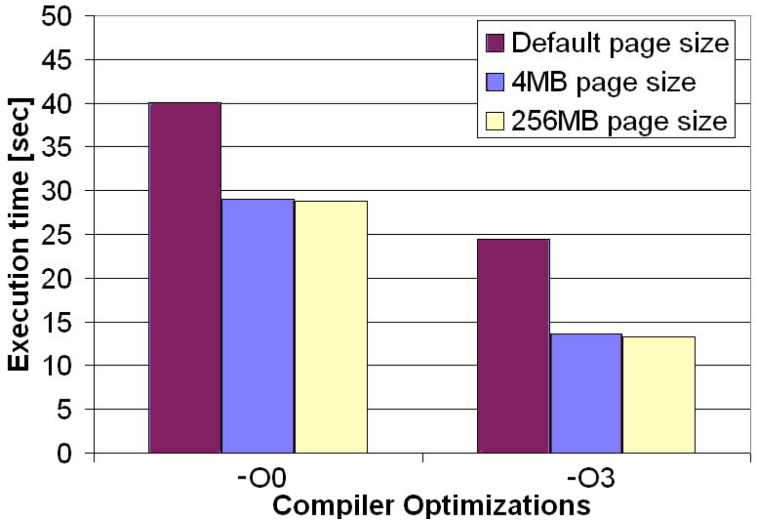


Figure 6.8. Effect of the page size on execution time under Solaris

and 4MB and 256MB page size causes an speedup up of 27.63% and 28.25% with respect to the case when application was using default page size. Speedup, with $-O3$ is 44.40% and 45.82% for 4MB and 256MB page size respectively. The configuration with 256MB heap page size provides the best results. In that case the overhead on the execution time is only 3.78% and 7.31% comparing to the execution in Netra DPS for $-O0$ and $-O3$ respectively. Using large memory page sizes causes the same absolute speedup (around 11 seconds) in both cases ($-O0$ and $-O3$).

Table 6.2 also shows that 4MB pages significantly reduces the number of data TLB misses (from 60 millions we had with the default page size) while increasing the page size to 256MB slightly additionally reduces number of data TLB misses.

6.2.4 Summary

In order to analyze memory address translation overhead, we execute memory intensive benchmark in Linux, Solaris, and Netra DPS. The initial results show significant performance drop when application runs in Linux and Solaris. Our analysis connects the performance drop to high number of TLB misses the application suffers when it is executed in Linux and Solaris. In following experiments, we manually increase the memory page size in Solaris and achieve the application performance close to the ones in Netra

DPS.

Our results show that that memory virtualization, even in the case of highly intensive memory benchmark, may introduces modest overhead if the size of memory page is properly established. However, currently it is responsibility of the user to set a proper page size to keep this overhead low. Given the importance of this feature, that may introduce high overhead, we think that the OS should provide a mechanism able to automatically set the best page size that fits application's memory behavior.

Chapter 7
Conclusions

The sources of performance overhead in operating systems have been deeply analyzed in the literature, with a strong focus on multichip computers. However, to our knowledge, this is the first work studying system overhead on a CMT chip. In our study, we compare execution time of several benchmarks on an UltraSPARC T1 processor running Linux and Solaris OSs, and Netra DPS low-overhead environment. In this chapter, we present the main conclusions of the thesis.

7.1 Netra DPS

In the study presented in the thesis, we use Netra DPS low-overhead environment. Netra DPS introduces less overhead than full-fledged operating systems because it provides less functionalities. Netra DPS does not have run-time process scheduler, interrupt handler nor daemons. It does not provide virtual memory abstraction and does not allow dynamic memory allocation. Because Netra DPS introduces almost no overhead, we use it as a baseline in order to quantify the overhead of Solaris and Linux.

7.2 OS process scheduler overhead

Our study shows that the process scheduler behavior in Linux and Solaris is significantly different.

In Linux, we detect the same process scheduler overhead in all strands. This is because of the fact that in Linux the process scheduler executes in every strand of the processor.

In Solaris, we detect different performance overhead depending on the strand a benchmark executes:

- When an application runs in *strand 0* we observe the highest overhead, regardless of the type of instructions the application executes.
- When the application runs in the same core with the timer interrupt handler, but on the strand different from *strand 0*, we also observe some smaller overhead the

intensity of which depends on the application's CPI (Cycles Per Instruction): the higher the CPI, the higher the overhead experimented by the application.

- We detect no timer interrupt overhead when applications execute on a core different than the one on which the timer interrupt handler runs.

The reason for this is that Solaris binds the timer interrupt handler to the *strand 0* of the logical domain, so no clock interrupt occurs in any strand different from *strand 0*.

Hence, high demanding application, sensitive to the overhead introduced by the timer interrupt, running in Solaris, should not run on the first core, definitely not in the *strand 0*. However, in the current version of Solaris, the scheduler does not take this into account when assigning a CPU to a process. Moreover, the scheduler may dynamically change the strand assigned to the application, so it is up to users to explicitly bind their applications to specific strands. In our experiments, when an application is not explicitly bound to any strand, Solaris schedules it on the *strand 0* for most of the execution, which leads to performance degradation.

7.3 Memory management overhead

The experiments running in Linux and Solaris experience significant slowdown because of memory address translation. On the other hand, since it does not provide virtual memory abstraction, Netra DPS introduces almost no memory management overhead.

Linux and Solaris, by default, use small memory pages. Since we use benchmarks that use large memory structures, the address translation requires a lot of memory map table entrances that do not fit in TLB. This causes significant number of TLB misses that directly affects application performance.

By increasing page size in Solaris, we significantly reduce the memory address translation overhead. Our result shows that memory virtualization overhead can be reduced to a small percentage by setting a page size that fits application memory requirements, while all the benefits of this service can be used by the user.

7.4 Parallel applications

The conclusions we obtain in the study come from single-threaded applications. Even so, they may be applied in scheduling of parallel applications running on a large number of processors where the slowdown suffered by any thread, for example due to a wrong scheduling decision, will likely affect the execution time of the entire application.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN- 2004-07739-C02-01, TIN-2007-60625, the HiPEAC European Network of Excellence and a Collaboration Agreement between Sun Microsystems and BSC. The authors wish to thank the reviewers for their comments, Jochen Behrens, Gunawan Ali-Santosa, Ariel Hendel and Arom Silverton from Sun for their technical support, and Bob Guarascio, also from Sun, for editing support.

Bibliography

[1] Xen 3.0 user manual. World Wide Web electronic publication.

[2] *OpenSPARC^{TM} T1 Microarchitecture Specification*, 2006.

[3] *UltraSPARC Architecture 2005*, 2006.

[4] *UltraSPARC T1^{TM} Supplement to the UltraSPARC Architecture 2005*, 2006.

[5] *Beginners Guide to LDoms: Understanding and Deploying Logical Domains*, 2007.

[6] *Logical Domains (LDoms) 1.0 Administration Guide*, 2007.

[7] *Netra Data Plane Software Suite 1.1 Getting Started Guide*, 2007.

[8] *Netra Data Plane Software Suite 1.1 Reference Manual*, 2007.

[9] *Netra Data Plane Software Suite 1.1 User's Guide*, 2007.

[10] World Wide Web electronic publication, 2009.

[11] Energy efficiency. World Wide Web electronic publication, 2009.

[12] Intel dual-core technology. World Wide Web electronic publication, 2009.

[13] Intel quad-core technology. World Wide Web electronic publication, 2009.

[14] T. E. Anderson, B. N. Bershad H. M. Levy, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[15] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly Media, Inc., 2006.

[16] Thomas D. Burd and Robert W. Brodersen. Design issues for dynamic voltage scaling. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 9–14, New York, NY, USA, 2000. ACM.

[17] I. B. Chen, A. Borg, and N. P. Jouppi. A simulation-based study of TLB performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 114–123, May 1992.

[18] D. W. Clark and J. S. Emer. Performance of the vax-11/780 translation buffers: Simulation and measurement. *ACM Transactions on Computer Systems*, February 1985.

[19] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology*, 2004.

[20] Mark D Hill and Alan J Smith. Experimental evaluation of on-chip microprocessor cache memories. Technical report, Berkeley, CA, USA, 1984.

[21] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 39–50, May 1993.

[22] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, July-August 1998.

[23] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[24] Jim Kahle. The cell processor architecture. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2005. IEEE Computer Society.

[25] G. B. Kandiraju and A. Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *ACM SIGMETRICS Performance Evaluation Review*, June 2002.

[26] Darren J. Kerbyson, Hank J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Michael Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of SC2001*, November 2001.

[27] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[28] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, February 2002.

[29] R. McDougall and J. Mauro. *Solaris^{TM} Internals*. Sun Microsystems Press, 2007.

[30] R. McDougall, J. Mauro, and B. Gregg. *Solaris^{TM} Performance and Tools*. Sun Microsystems Press, 2007.

[31] J. Mogul. Big memories on the desktop. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 110–115, October 1993.

[32] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[33] T. Romer, W. Ohlrich, A. Karlin, and B. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 176–187, May 1995.

[34] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[35] Amit Singh. An introduction to virtualization. World Wide Web electronic publication, 2006.

[36] A. Srwastava and A. Eustace. Atom: A system for budding customized program analysis tools. In *Proceeding of the 1994 ACM Symposium on Programming Languages Design and lmplementation ACM*, 1994.

[37] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 171–182, October 1994.

[38] M. Talluri, S. Kong, M. D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th annual international symposium on Computer architecture (ISCA '92)*, pages 415–424, May 1992.

[39] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Experimental computer science on Experimental computer science*, pages 171–182, 2007.

[40] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, 2005.

[41] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[42] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandéz, and M. Valero. Analysis of system overhead on parallel computers. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 305–316, 2007.

[43] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandéz, and M. Valero. Measuring the Performance of Multithreaded Processors. In *SPEC Benchmark Workshop*, 2007.